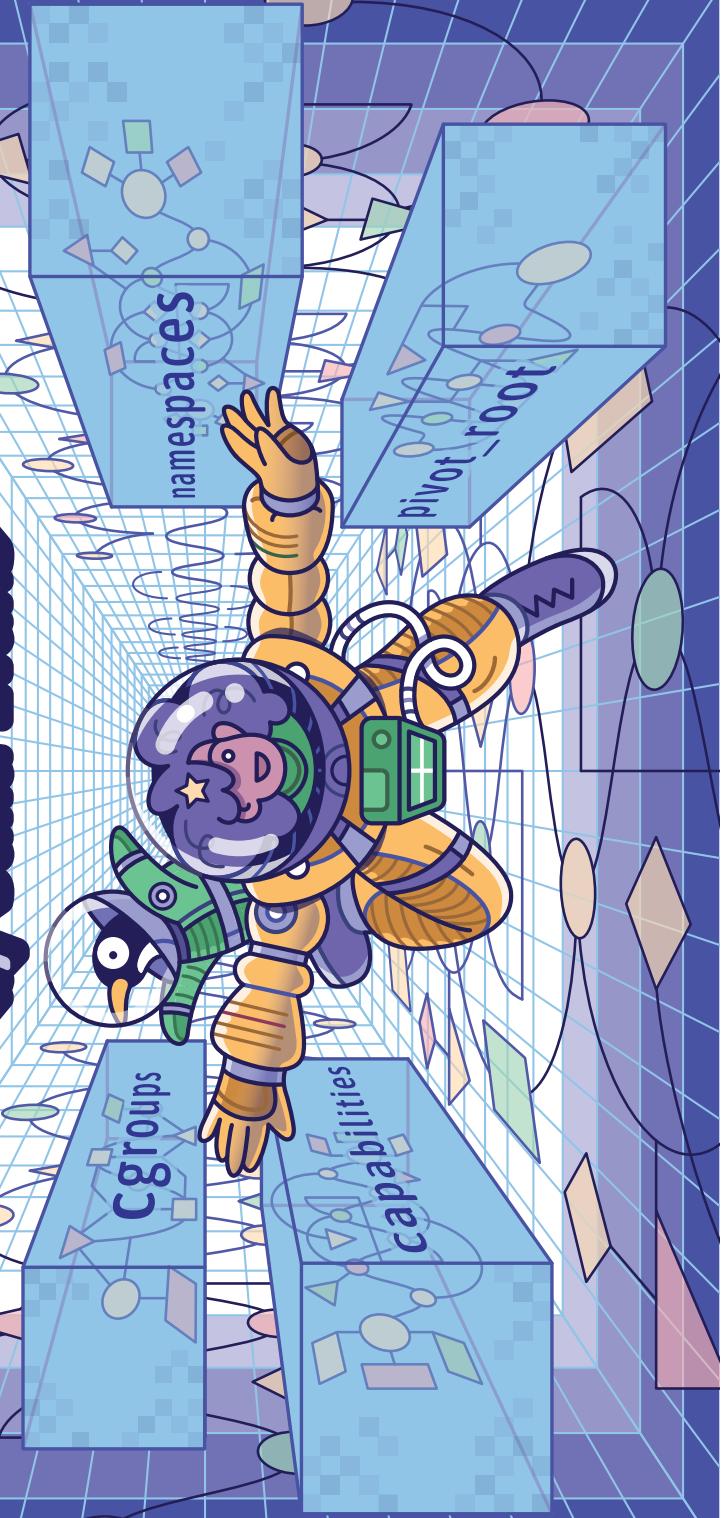


HOW CONTAINERS WORK

by Julia Evans



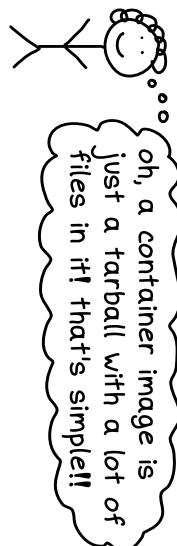
* wizardries.com
more at
@this?

why this zine?

When I started using containers I was SO CONFUSED.



So I decided to learn how they work under the hood!



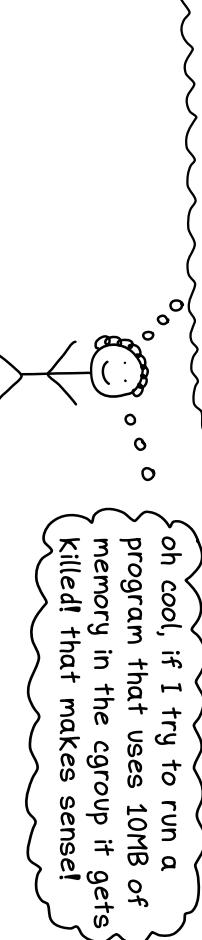
Now I feel confident that I can solve basically any problem with containers because I understand how they work. I hope that after reading this zine, you'll feel like that too.



♥ thanks for reading ♥

I did a bunch of the research for this zine by reading the man pages. But, much more importantly, I experimented -- a lot!

(let's see, what happens if I create a cgroup with a memory limit of 5MB?)



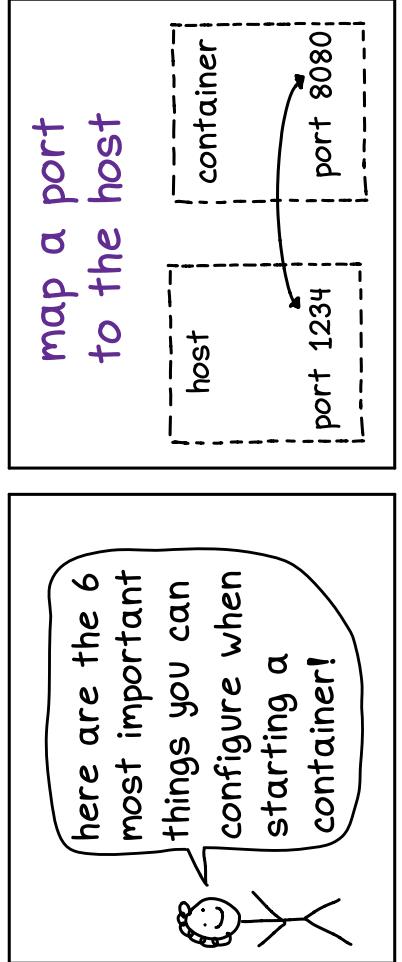
So, if you have access to a Linux machine, try things out! Mount an overlay filesystem! Create a namespace! See what happens!

[credits](#)

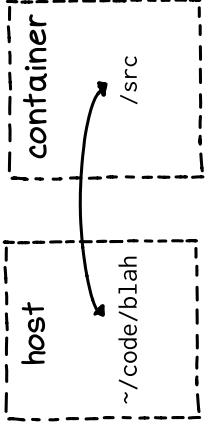
Cover art: Vladimir Kašiković
Editing: Dolly Lanuza, Kamal Marhubi

configuration options

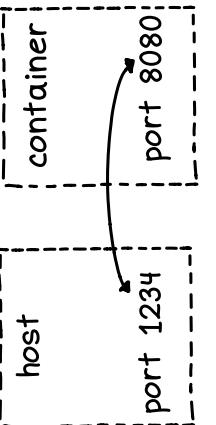
22



mount directories from the host



map a port to the host



set capabilities

use the host network namespace

add seccomp-bpf filters

usually the default is to use a new network namespace!

set memory and CPU limits

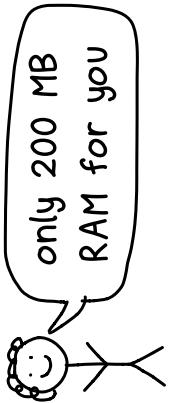


table of contents

| | | | |
|--|----|-------------------------------|----|
| why containers? | 4 | cgroups | 13 |
| the big idea: include EVERY dependency | 5 | namespaces | 14 |
| containers aren't magic | 6 | how to make a namespace | 15 |
| containers = processes | 7 | PID namespaces | 16 |
| container kernel features | 8 | user namespaces | 17 |
| pivot_root | 9 | network namespaces | 18 |
| layers | 10 | container IP addresses | 19 |
| overlay filesystems | 11 | capabilities | 20 |
| container registries | 12 | seccomp-BPF | 21 |
| | | configuration options | 22 |

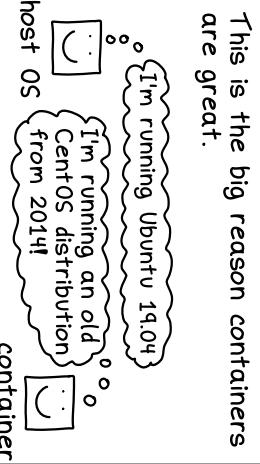
why containers?

there's a lot of container hype



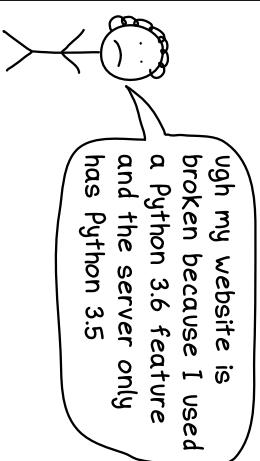
Here are 2 problems they solve...

containers have their own filesystem



This is the big reason containers are great.

problem: building software is annoying
\$./configure
make all
ERROR: you have version 2.1.1 and you need at least 2.2.4



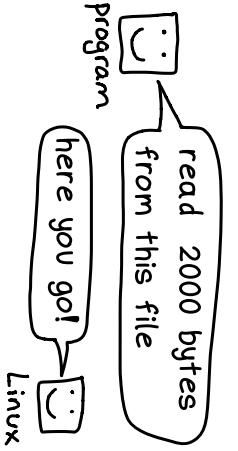
problem: deploying software is annoying too

solution: package all dependencies in a container ★
Many CI systems use containers.
I ran the container and the build worked RIGHT AWAY?? Is that allowed??

seccomp-bpf

21

all programs use system calls



rarely-used system calls can help an attacker
request_key reboot process_vm_ready
read memory from another process

seccomp-BPF lets you run a function before every system call



the function decides if syscall is allowed

example function:

```
if name in allowed_list {  
    return true;  
} else {  
    this means the syscall doesn't happen!  
}
```

Docker blocks dozens of syscalls by default

most programs don't need those system calls so I told Linux to block them for you!

2 ways to block scary system calls

1. limit the container's capabilities
2. set a seccomp-bpf whitelist

You should do both!

capabilities

20

we think of root as being all-powerful...

edit any file
change network config
spy on any program's memory

... but actually to do "root" things, a process needs the right ★ capabilities★

there are dozens of capabilities

\$ man capabilities
explains all of them
but let's go over 2 important ones!

CAP_SYS_ADMIN

lets you do a LOT of things.
avoid giving this if you can!

by default containers have limited capabilities

can I call process_vm_readv?

nope! you'd need CAP_SYS_PTRACE for that!

CAP_NET_ADMIN

allow changing network settings

\$ getpcaps PID

print capabilities
that PID has

getcap / setcap

system calls:
get and set capabilities!

5

the big idea: include EVERY dependency

containers package EVERY dependency together

to make sure this program will run on your laptop, I'm going to send you every single file you need

a container image is a tarball of a filesystem

Here's what's in a typical Rails app's container:

your app's code
Ruby interpreter
Ruby gems
libc + other system libraries
Ubuntu base OS

how images are built

0. start with a base OS
1. install program + dependencies
2. configure it how you want
3. make a tarball of the WHOLE FILESYSTEM

this is what docker build` does!

running an image

1. download the tarball
2. unpack it into a directory
3. run a program and pretend that directory is its whole filesystem

I can set up a Postgres test database in like 5 seconds! wow!

containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!
 (download this script at bit.ly/containers-arent-magic)

It only runs on Linux because these features are all Linux-only.

```
wget bit.ly/fish-container -O fish.tar
mkdir container-root; cd container-root
tar -xf ./fish.tar
# 1. download the image

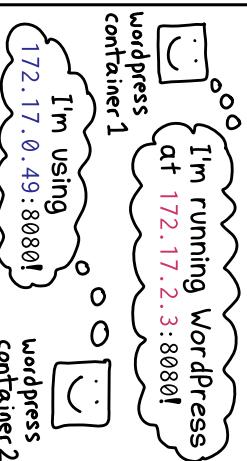
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)" # 2. unpack image into a directory
cgcreate -g "cpu,cpuacct,memory:$cgroup_id" # 3. generate random cgroup name
cgset -r cpu.shares=512 "$cgroup_id" # 4. make a cgroup & set CPU/memory limits
cgset -r memory.limit_in_bytes=1000000000 \ # $cgroup_id"
# 5. use the cgroup

cgexec -g "cpu,cpuacct,memory:$cgroup_id" \
unshare -fmuipn --mount-proc \
chroot "$PWD" \
/bin/sh -c "
# 6. make + use some namespaces
# 7. change root directory
/bin/mount -t proc proc /proc &
# 8. use the right /proc
hostname container-fun-times &&
# 9. change the hostname
/usr/bin/fish" # 10. finally, start fish!
```

container IP addresses

19

containers often get their own IP address



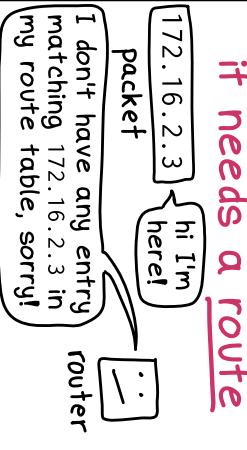
containers use private IP addresses

| | |
|-------------|---|
| 192.168.*.* | } reserved for private networks (RFC 1918) |
| 10.*.*.* | |
| 172.16.*.* | |

-> 172.32.*.*

This is because they're not directly on the public internet

for a packet to get to the right place, it needs a route



I don't have any entry matching 172.16.2.3 in my route table, sorry!

inside the same computer, you'll have the right routes

same computer:

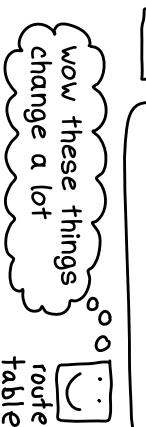
```
$ curl 172.16.2.3:8080
<html>...</html>
```

different computer:

```
$ curl 172.16.2.3:8080
... no reply ...
```

distributing the right routes is complicated

a new container started, 10.2.73.4 should go to X computer now



cloud providers have systems to make container IPs work

In AWS this is called an "elastic network interface"

network namespaces

18

network namespaces are kinda confusing

- what does it MEAN for a process to have its own network??

namespaces usually have 2 interfaces (+ sometimes more)

- the loopback interface (127.0.0.1/8, for connections inside the namespace)
- another interface (for connections from outside)

every server listens on a port and network interface(s)
0.0.0.0:8080 means "port 8080 on every network interface in my namespace"

127.0.0.1 stays inside your namespace

- I'm listening on 127.0.0.1
- that's fine but nobody outside your network namespace will be able to make requests to you!

your physical network card is in the host network namespace

The diagram shows a dashed box labeled 'host network namespace' containing four smaller boxes, each labeled 'container'. Each container has a small network card icon next to it, labeled '192.168.1.149'.

requests from other computers

other namespaces are connected to the host namespace with a bridge

The diagram shows a dashed box labeled 'host network namespace' connected by lines to four smaller boxes, each labeled 'container'. Each container has a small network card icon next to it.

containers = processes

7

a container is a group of Linux processes

- on a Mac, all your containers are actually running in a Linux virtual machine

inside the container

```
$ ps aux | grep top
USER PID START COMMAND
root 23540 20:55 top
bork 23546 20:57 top
```

these two are the same process!

container processes can do anything a normal process can ...

... but usually they have restrictions

- different PID
- different namespace
- different root directory
- limited memory (limit)
- limited capabilities
- not allowed to run some system calls

I want my container to do X Y Z W!

sure! your computer, your rules!

NO, you can't have more memory!

on the next page we'll list all the kernel features that make this work!

Container kernel features

Containers use these Linux Kernel features

"container" doesn't have a clear definition, but Docker containers use all of these features.

✿ pivot_root ✿

set a process's root directory to a directory with the contents of the container image

Linux
only 500 MB of RAM for you!

★ cgroups ★

limit memory/CPU usage for a group of processes

✿ namespaces ✿

allow processes to have their own:

- network → mounts
- PIDs → users
- hostname + more

✿ seccomp-bpf ✿

security: prevent dangerous system calls

★ overlay filesystems ★

this is what makes layers work! Sharing layers saves disk space & helps containers start faster

User namespaces

17

User namespaces are a security feature...

I'd like root in the container to be totally unprivileged

you want a user namespace!

... but not all container runtimes use them

same user!

"root" doesn't always have admin access

I'm root so I can do ANYTHING right?

actually you have limited capabilities so mostly you can just access files owned by root!

In a user namespace, UIDs are mapped to host UIDs

I'm running as UID 0
oh, that's mapped to 12345
Linux

The mapping is in /proc/self/uid_map

unmapped users show up as "nobody"

create user namespace
\$ unshare --user bash

\$ ls -l /usr/bin

.. nobody nogroup apropos

.. nobody nogroup apt

(these are "actually" owned by root but we didn't map any users)

how to find out if you have a separate user namespace

compare the results of \$ ls /proc/PID/ns between a container process and a host process.

PID namespaces

16

the same process has different PIDs in different namespaces

PID in host PID in container
23512 1
23513 4
23518 12
PID 1 is special

PID namespaces are in a tree

host PID namespace
(the root)
↓
child child child
Often the tree is just 1 level deep (every child is a container)

you can see processes in child PID namespaces

aw! look at all those containers running!
o o o



if PID 1 exits, everyone gets killed

PID 1 ok I'm done!
I'm kill -9'ing everyone else in this PID namespace IMMEDIATELY



do you WANT everyone to die? I'm not gonna let you do that



killing PID 1 accidentally would be bad



kill 1



do you WANT everyone to die? I'm not gonna let you do that



pivot_root

a container image is a tarball of a filesystem

(or several tarballs: 1 per layer)

if someone sends me a tarball of their filesystem, how do I use that though?



chroot: change a process's root directory

If you chroot to /fake/root, when it opens the file /usr/bin/redis it'll get /fake/root/usr/bin/redis instead.

You can "run" a container just by using chroot, like this:

```
$ mkdir redis; cd redis  
$ tar -xzf redis.tar  
$ chroot $PWD /usr/bin/redis  
# done! redis is running!
```

programs can break out of a chroot

chroot

redis container directory

all these files are still there! A root process can access them if it wants.

Containers use pivot_root instead of chroot.

to have a "container" you need more than pivot_root

pivot_root alone won't let you:

- set CPU/memory limits
- hide other running processes
- use the same port as another process
- restrict dangerous system calls

layers

different images have similar files



every layer has an ID

usually the ID is a sha256 hash of the layer's contents

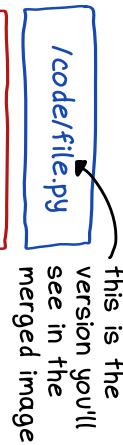
example: 8e99fae2...

reusing layers saves disk space



exact same files on disk!

if a file is in 2 layers, you'll see the version from the top layer



by default, writes go to a temporary layer

[temp layer]

these files might be deleted after the container exits

To keep your changes, write to a directory that's mounted from outside the container

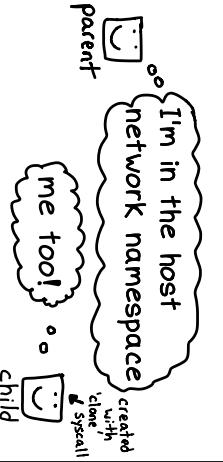
a layer is a directory

```
$ ls 8891378eb*
bin/ home/ mnt/ run/ tmp/
boot/ lib/ opt sbin/ usr/
dev/ lib64/ proc/ srv/ var/
etc/ media/ root/ sys/
files in an ubuntu:18.04 layer
```

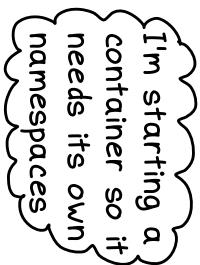
how to make a namespace

15

processes use their parent's namespaces by default



but you can switch namespaces at any time



command line tools

```
$ unshare --net COMMAND
run COMMAND in a new network namespace
```

```
$ sudo lsn
list all namespaces
```

```
$ nsenter -t PID --all COMMAND
run COMMAND in the same namespaces as PID
```

each namespace type has a

man page

\$ man network_namespaces

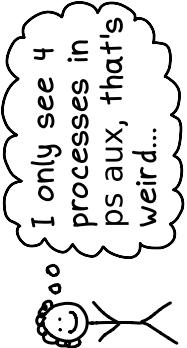
... physical network device can live in exactly one network namespace.

- ★ **namespace system calls**
- ★ **clone** ★ lets you create new namespaces for a child process
- ★ **make a new process**
- ★ **unshare** ★ make + use a namespace
- ★ **setsns** ★ use an existing namespace

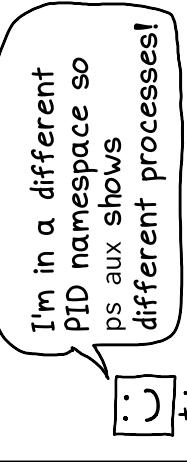
namespaces

14

inside a container,
things look different



why things look different:
↳ namespaces:



every process has
7 namespaces

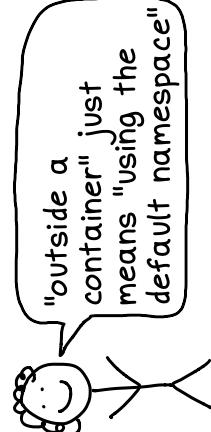
```
$ lsns -p 273
      PID          NS TYPE
4026531835 cgroup
4026531836 pid
4026531837 user
4026531838 uts
4026531839 ipc
4026531840 mnt
4026532009 net
```

↑ namespace ID

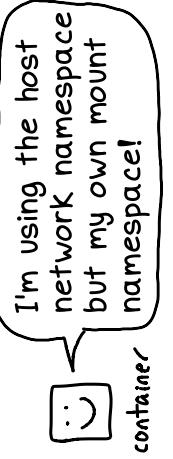
you can also see a process's namespace with:

```
$ ls -l /proc/273/ns
```

there's a default ("host" namespace)



processes can have
any combination
of namespaces

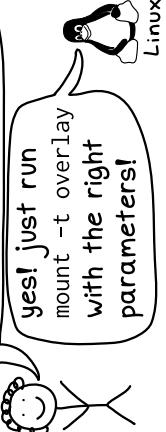


Overlay filesystems

11

how layers work:
mount -t overlay

can you combine these 37
layers into one filesystem?



mount -t overlay
has 4 parameters

lowerdir: list of read-only directories
upperdir: directory where writes should go

workdir:

empty directory for internal use

target:

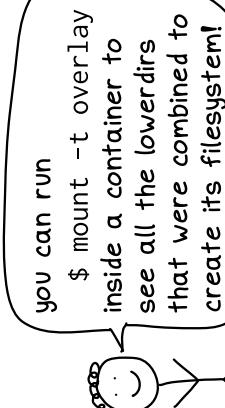
the merged result

upperdir:
where all writes go

when you create, change, or
delete a file, it's recorded in
the upperdir.

usually this starts out empty
and is deleted when the
container exits

lowerdir:
the layers. read only.



here's an example!

```
$ mount -t overlay overlay -o
  lowerdir=/lower,upperdir=/upper,workdir=/work /merged
```

```
$ ls /upper
cat.txt dog.txt
$ ls /lower
dog.txt bird.txt
$ ls /merged
cat.txt dog.txt bird.txt
```

the merged version of dog.txt is
the one from the upper directory

container registries

sharing container images is useful



there are **public** container registries...



a registry is a server that serves images

images have an **ID** like "leff92"
and sometimes a **tag**
like "18.04" or "latest"

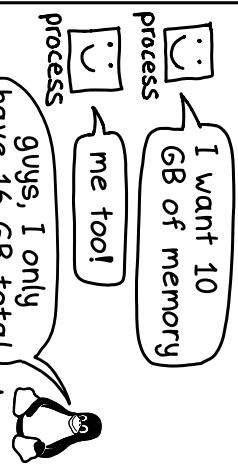
... and **private** registries



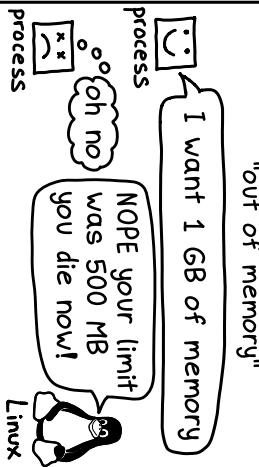
be **careful** where your container images come from



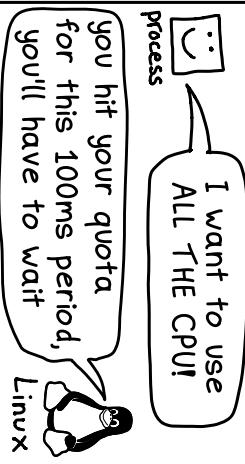
processes can use a lot of memory



use too much memory: get OOM Killed



use too much CPU: get slowed down



cgroups track memory & CPU usage



cgroups

cgroups have memory/CPU limits

